

字节码的编译原理

一、javac

字节码的编译过程，需要经过下面的步骤：

- 词法解析。
- 语法解析。
- 语义解析。
- 字节码生成。

1.1 语义解析

在语义解析时，会经历一些步骤，如：

- 为没有构造方法的类型，添加缺省的无参构造方法；
- 检查任何类型的变量在使用前是否已经初始化；
- 检查变量类型是否与值匹配；
- 将string类型的常量进行合并处理；
- 检查代码中的所有偶操作语句是否可达；
- 异常检查；
- 解除java语法糖

1.2 生成字节码

在经过词法分析、语法分析和语义分析步骤之后，解析出来的语法树已经非常完善了，javac编译的最后任务就是调用com.sun.tools.javac.jvm.Gen类将这个语法树编译为java字节码文件。JVM的架构模式，是基于栈的，在jvm中的所有操作都要经过入栈和出栈来完成。

二、javap工具分析字节码

如下是javap的帮助文档：

```

C:\Users\Administrator>javap --help
用法: javap <options> <classes>
其中, 可能的选项包括:
- help  --help  -?      输出此用法消息
- version      版本信息
- v  -verbose    输出附加信息
- l            输出行号和本地变量表
- public       仅显示公共类和成员
- protected    显示受保护的/公共类和成员
- package      显示程序包/受保护的/公共类
                  和成员 (默认)
- p  -private   显示所有类和成员
- c            对代码进行反汇编
- s            输出内部类型签名
- sysinfo      显示正在处理的类的
                  系统信息 (路径, 大小, 日期, MD5 散列)
- constants    显示最终常量
- classpath <path> 指定查找用户类文件的位置
- cp <path>      指定查找用户类文件的位置
- bootclasspath <path> 覆盖引导类文件的位置

C:\Users\Administrator>_

```

为了验证语义分析步骤中的语义解析器是否会为没有显示指定构造方法的类型动态添加一个无参缺省构造方法, 先使用示例代码DemoTest1.java来验证。

源代码:

```

public class DemoTest1 {
    public static void main(String[] args) {
        final String STR = "Hello " + "World";
        System.out.println(STR);
    }
}

```

编译后的class文本内容:

DemoTest1.class x																
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	CA	FE	BA	BE	00	00	00	34	00	1D	0A	00	06	00	0F	09 ; 漱壕...4.....
00000010h:	00	10	00	11	08	00	12	0A	00	13	00	14	07	00	15	07 ;
00000020h:	00	16	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29 ;<init>...()
00000030h:	56	01	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E ; V...Code...LineN
00000040h:	75	6D	62	65	72	54	61	62	6C	65	01	00	04	6D	61	69 ; umberTable...mai
00000050h:	6E	01	00	16	28	5B	4C	6A	61	76	61	2F	6C	61	6E	67 ; n...([Ljava/lang
00000060h:	2F	53	74	72	69	6E	67	3B	29	56	01	00	0A	53	6F	75 ; /String;)V...Sou
00000070h:	72	63	65	46	69	6C	65	01	00	0E	44	65	6D	6F	54	65 ; rceFile...DemoTe
00000080h:	73	74	31	2E	6A	61	76	61	0C	00	07	00	08	07	00	17 ; st1.java.....
00000090h:	0C	00	18	00	19	01	00	0B	48	65	6C	6C	6F	20	57	6F ;Hello Wo
000000a0h:	72	6C	64	07	00	1A	0C	00	1B	00	1C	01	00	09	44	65 ; rld.....De
000000b0h:	6D	6F	54	65	73	74	31	01	00	10	6A	61	76	61	2F	6C ; moTest1...java/l
000000c0h:	61	6E	67	2F	4F	62	6A	65	63	74	01	00	10	6A	61	76 ; ang/Object...jav
000000d0h:	61	2F	6C	61	6E	67	2F	53	79	73	74	65	6D	01	00	03 ; a/lang/System...
000000e0h:	6F	75	74	01	00	15	4C	6A	61	76	61	2F	69	6F	2F	50 ; out...Ljava/io/P
000000f0h:	72	69	6E	74	53	74	72	65	61	6D	3B	01	00	13	6A	61 ; rintStream;...ja
00000100h:	76	61	2F	69	6F	2F	50	72	69	6E	74	53	74	72	65	61 ; va/io/PrintStrea
00000110h:	6D	01	00	07	70	72	69	6E	74	6C	6E	01	00	15	28	4C ; m...println...(L
00000120h:	6A	61	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	67 ; java/lang/String
00000130h:	3B	29	56	00	21	00	05	00	06	00	00	00	00	02	00	00 ; ;)V.!.....
00000140h:	01	00	07	00	08	00	01	00	09	00	00	00	1D	00	01	00 ;
00000150h:	01	00	00	00	05	2A	B7	00	01	B1	00	00	00	01	00	0A ;*?.?.....
00000160h:	00	00	00	06	00	01	00	00	00	01	00	09	00	0B	00	0C ;
00000170h:	00	01	00	09	00	00	00	25	00	02	00	02	00	00	00	09 ;%.....
00000180h:	B2	00	02	12	03	B6	00	04	B1	00	00	00	01	00	0A	00 ; ?...??.?.....
00000190h:	00	00	0A	00	02	00	00	00	04	00	08	00	05	00	01	00 ;
000001a0h:	0D	00	00	00	02	00	0E									;

我们在用javap反编译一下，看得到什么：

```
E:\svnrepo\learning-projects\learning-jvm\codes>javap DemoTest1.class
Compiled from "DemoTest1.java"
public class DemoTest1 {
    public DemoTest1();
    public static void main(java.lang.String[]);
}

E:\svnrepo\learning-projects\learning-jvm\codes>
```

可以看到多了一个无参构造方法出现，因此语义分析的步骤中，验证成功。

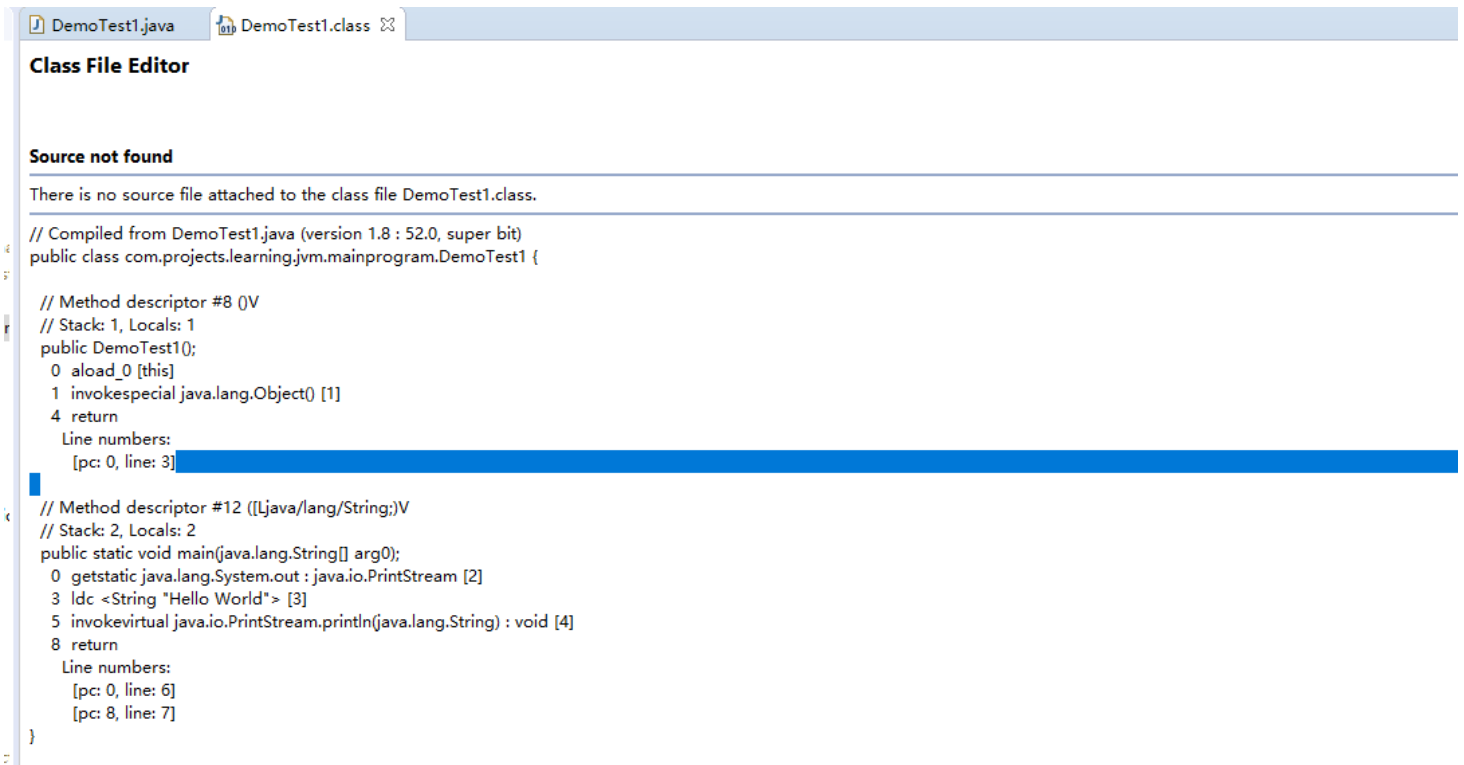
如果用Eclipse或者其他UI形式的反编译工具打开这个class文件，则大概看到如下内容：

```
// Compiled from DemoTest1.java (version 1.8 : 52.0, super bit)
public class com.projects.learning.jvm.mainprogram.DemoTest1 {

    // Method descriptor #8 ()V
    // Stack: 1, Locals: 1
    public DemoTest1();
        0  aload_0 [this]
        1  invokespecial java.lang.Object() [1]
        4  return
        Line numbers:
            [pc: 0, line: 3]

    // Method descriptor #12 ([Ljava/lang/String;)V
    // Stack: 2, Locals: 2
    public static void main(java.lang.String[] arg0);
        0  getstatic java.lang.System.out : java.io.PrintStream [2]
        3  ldc <String "Hello World"> [3]
        5  invokevirtual java.io.PrintStream.println(java.lang.String) : void [4]
        8  return
        Line numbers:
            [pc: 0, line: 6]
            [pc: 8, line: 7]
}
```

也同樣可以看到新增了一個構造方法。



Class File Editor

Source not found

There is no source file attached to the class file DemoTest1.class.

```
// Compiled from DemoTest1.java (version 1.8 : 52.0, super bit)
public class com.projects.learning.jvm.mainprogram.DemoTest1 {

    // Method descriptor #8 ()V
    // Stack: 1, Locals: 1
    public DemoTest1();
        0  aload_0 [this]
        1  invokespecial java.lang.Object() [1]
        4  return
        Line numbers:
            [pc: 0, line: 3]

    // Method descriptor #12 ([Ljava/lang/String;)V
    // Stack: 2, Locals: 2
    public static void main(java.lang.String[] arg0);
        0  getstatic java.lang.System.out : java.io.PrintStream [2]
        3  ldc <String "Hello World"> [3]
        5  invokevirtual java.io.PrintStream.println(java.lang.String) : void [4]
        8  return
        Line numbers:
            [pc: 0, line: 6]
            [pc: 8, line: 7]
}
```

三、GCJ

GCJ（GNU Compiler for the Java Programming Language）编译器可以将java源码直接编译为机器码，一旦GCJ编译器编译出机器码指令之后，java程序就可以脱离jvm环境独立运行。

两款著名的"元循环"虚拟机(Mete-Circular JVM),分别是JavalnJava和Maxine VM。所谓"元循环"指的是使用java语言自身编写的虚拟机去运行java程序。

尽管可以这么做，但是还是会有缺陷。

- 一旦java被编译成机器码，那么就失去了"write once , run anywhere"的实现
- 编译为可执行程序之后的java程序体积高达几十兆
- gcj只会针对java的基础类库提供支持，其他的三方库,GCJ无能为力

所以最好还是由HotSpot中的jit编译器去负责本地机器指令的动态编译工作。